Rapport de projet informatique : INSAtram

Alix ANNERAUD - Myriem ABID - Hugo LASCOUTS

23 décembre 2022

Table des matières

1 Introduction.

1.1 Présentation

Dans le cadre du projet informatique que nous réalisons suite aux cours de I1 et I2, nous avons décidé de développer un jeu en Pascal. Le nom du projet est « INSAtram » où « Tram » fait référence au moyen de transport urbain. Le jeu consiste en la construction et gestion d'un réseau de transport ferroviaire. C'est pourquoi l'identité graphique du jeu s'apparente aux cartes de réseau disponibles dans les transports en commun. Le concept n'est pas original puisqu'il est inspiré du jeu « Mini Metro » de l'éditeur « Dinosaur Polo Club ». L'objectif du joueur est d'acheminer des passagers vers leur destination, et ce, de manière optimale. Plusieurs stations, qu'il faudra relier entre elles, apparaissent au cours de la partie. Les passagers apparaissent à côté de leur station de départ et empruntent la ou les lignes qui desservent leur station an d'atteindre leur destination. Ce projet était à réaliser en groupe, le nôtre était composé de 3 membres. Par conséquent, en plus de la mobilisation des connaissances acquises en informatique, il était nécessaire de prendre en compte le travail de groupe et donc la communication ainsi que la répartition des tâches. Cette dimension, bien qu'indispensable, a rajouté une difficulté à la planification du projet. En effet, jusqu'ici, nous étions habitués à travailler de manière individuelle et sans trop se soucier d'établir un plan de projet de A à Z. Nous présentons dans ce rapport notre façon d'appréhender ce nouvel exercice. Le répertoire du projet est disponible à l'adresse : https ://github.com/AlixANNERAUD/INSAtram.

1.2 Cahier des charges initial

Voici un rappel du cahier des charges. Les parties soulignées sont les fonctionnalités optionnelles et leur réalisation est faite suivant l'avancement du projet :

- **Interface graphique :** L'interface graphique comportera les éléments suivants :
  - Un menu de démarrage permettant de démarrer une partie.
  - Un terrain de jeu contenant tous les éléments de la partie (score, temps, inventaire).
  - Un bilan à la fin de la partie (au game over) affichant le score du joueur.
  - La présence de couleurs tout en gardant une interface épurée.
- **Temps :** Organisé en jours et semaines (jour : ~20 secondes, 1 semaine : ~140 secondes). Il détermine en partie le score final du joueur. Il est possible de mettre en pause et de reprendre le temps.
- **Son :** Le jeu comprend une musique de fond apaisante pour mettre le joueur dans l'ambiance du jeu.
- **Objets :** Ce sont les éléments avec lesquels le joueur peut interagir. Tous les objets sont présents en début de partie en quantité limitée et peuvent être obtenus à la fin de chaque semaine. Les objets seront les suivants :
  - **Locomotives :** Elles effectuent des allers-retours sur les lignes afin d'acheminer les voyageurs de station en station.
  - **Lignes :** Ce sont les infrastructures permettant aux véhicules de circuler de station en station. Le joueur les dessine et peut les modifier au cours de la partie.
  - **Wagons :** Ce sont des véhicules pouvant être ajoutés à une locomotive afin d'augmenter le nombre de passagers qu'elle peut transporter en une seule fois.
  - **Tunnels :** Ce sont les objets permettant à une ligne de traverser une rivière.
- **Entités :** Ce sont les éléments qui sont disposés sur le terrain de jeu par le programme. Les entités seront les suivantes :
  - **Stations :** Elles sont de différentes formes géométriques et sont générées de manière aléatoire. Elles apparaissent à des instants t déterminés par une fonction.
  - **Passagers :** Ils sont également de différentes formes géométriques (correspondant à leur station d'arrivée) et ils apparaissent à côté des stations à des instants t déterminés par une fonction. Ces passagers disparaissent une fois arrivés à destination. L'algorithme de Dijkstra est utilisé afin de calculer les itinéraires des passagers.
  - **Rivières :** Les différentes cartes comprennent une ou plusieurs rivières qui font office d'obstacles naturels. Elles doivent être contournées à l'aide d'un tunnel.
- **Difficulté :** La difficulté du jeu réside dans le fait de devoir fluidifier au maximum le trafic et d'adapter le réseau en fonction des nouvelles stations qui apparaissent, le flux de passagers ainsi que des différents aléas

— **Fin de partie :** La partie prend fin une fois qu'une station est engorgée (quand plus de 6 passagers s'y trouvent) pendant 40 secondes. Un chronomètre apparaît à côté de la/des stations surchargée(s) pour indiquer le temps restant au joueur. Si ce temps est dépassé alors que la station est encore encombrée, le joueur perd.

2 Structure

Au début du projet, nous avons essayé d'être le plus rigoureux possible lorsque l'on a déterminé sa structure. Cependant, certaines parties ne pouvaient être prédites lorsque le projet était encore au stade de simple idée. Nous avons donc dû apporter des modifications à notre analyse descendante initiale, la version qui a été mise à jour se trouve en annexe du rapport. Les fonctions et procédures étant nombreuses dans notre programme, notre analyse descendante ne les englobe pas toutes et montre seulement les fonctions les plus importantes. Nous avons également veillé à séparer le code en plusieurs unités et à laisser des commentaires dans le but d'avoir un code lisible et compréhensible. Certains « writeln » qui ne sont pas utiles à l'exécution du code ont été ajoutés afin d'afficher du texte dans le terminal pour faciliter la compréhension du déroulement de la partie.

2.1 Nommage

Afin que le code soit structuré et facile à comprendre, nous avons adopté une convention d'écriture qui suit les règles suivantes :

— Langue : l'écriture du code en anglais nous semble assez naturelle, pour des raisons d'homogénéité avec le langage de programmation lui-même. Cependant, afin de faciliter la compréhension du code lors de la correction, nous avons utilisé du français pour les commentaires. Ces derniers suivent une certaine structure. Pour un commentaire faisant office de titre :

```
1 // - Section
2
3 // -- Sous-section
4
5 // Commentaire standard.
6 Procedure Station_Render();
```

— Constantes : Les constantes ont en général leur nom qui commence par l'identifiant de l'élément concerné. Cependant, dans le cas des chemins des fichiers, les constantes commencent par « Path_ » suivi du type de l'élément concerné (Image, Police, Musique).

```
1 Const Station_Overfill_Timer = 20;
2
3 Const Path_Image_Station_Circle = '/Resources/Images/Station_Circle.png'
```

— Types : Tout type porte d'abord comme identifiant Type_ suivi de l'identifiant du type. Cet identifiant doit être le plus court et le plus explicite possible. Par exemple, la structure contenant une station s'appelle Type_Station . Dans le cas d'un type pointeur, il suffit de rajouter _Pointer au nom du type de base :

```
1 Type Type_Station = Record
2     ...
3 End;
4
5 Type Type_Station_Pointer = ^Type_Station;
```

— Fonctions et procédures : Les fonctions et procédures portent généralement le nom de l'objet sur lequel elles agissent ou de l'unité concernée, suivi d'un verbe (en général : Get , Set , Refresh , Render ...) indiquant l'action effectuée et est éventuellement suivi de précisions sur l'action :

```
1 // Procedure rafraichissant les graphismes
2 Procedure Graphics_Refresh();
3
4 // Procedure qui rend graphiquement une station
5 Procedure Station_Render();
```

2.2 Structure générale

Le projet est structuré en plusieurs fichiers de la manière suivante (voir annexes) :

- **Unit_Types :** Unité contenant tout les types
- **Unit_Constants :** Unité contenant toutes les constantes du jeu.
- **Unit_Common :** Unité contenant toutes les fonctions « élémentaires » permettant de simplifier le développement dans les autres unités.
- **Unit_Mouse :** Unité regroupant toutes les fonctions responsables de la prise en charge de la souris.
- **Unit_Animation :** Unité regroupant toutes les fonctions responsables des animations (déplacement des trains...).
- **Unit_Graphics :** Unité regroupant toutes les fonctions relatives aux graphismes.
- **Unit_Logic :** Unité regroupant toute la logique du jeu. C'est l'unité « mère » du jeu, celle qui est directement inclue dans le programme principal : « INSAtram.pas ».

3 Organisation

3.1 Outils utilisés

Ce projet étant un projet plus complexe que ce que nous avons eu à programmer jusqu'ici, nous avons fait appel à plusieurs outils afin de répondre au mieux à ses besoins. Voici la liste des outils utilisés pour le développement du jeu :

- Visual Studio Code : Un éditeur de code, open source et gratuit, très extensible et personnalisable.
- Git : Logiciel de gestion de répertoire de code, open source et performant.
- GitHub : Plateforme en ligne d'hébergement de répertoire de code Git.
- Live Share : Une extension de Visual Studio Code permettant une collaboration en direct (à la manière d'un Google Docs).
- Ptop : Le formateur de code fourni avec le Free Pascal Compiler.
- Inkscape : Logiciel de dessin vectoriel utilisé pour la création des ressources graphiques du jeu.
- Mermaid : Outil de création dynamique de diagrammes et de graphiques utilisé pour la création de l'analyse descendante.
- LyX : Logiciel libre d'édition de documents techniques. Nous l'avons utilisé pour la rédaction du présent rapport.

3.2 Répartition

Afin d'avoir une répartition du travail la plus homogène possible, nous avons adopté une méthode similaire à la méthode « Agile ». C'est-à-dire que nous avons essayé de diviser le cahier des charges en tâches élémentaires. Ces tâches étaient disposées sur un tableau virtuel dans le but de les rendre accessibles à chaque membre du groupe. Puis, chaque personne pouvait récupérer une tâche dès que la précédente était implémentée, testée et validée par les autres. Cette méthode permettait alors à chacun d'explorer différentes parties du développement et d'avoir une vision d'ensemble quant à l'avancement du projet. Cependant, certaines tâches nécessitaient des calculs et raisonnements complexes, dans ces cas-là, nous étions au moins deux à travailler dessus. De ce fait, il serait difficile d'établir une liste exhaustive des tâches effectuées par chaque membre du groupe.

Il était parfois difficile de faire correspondre nos emplois du temps respectifs. Alors, pour ne pas avoir à être systématiquement en présentiel pour pouvoir travailler ensemble sur le projet, nous avons utilisé des outils tels que Git et GitHub pour la gestion du répertoire de code et LiveShare afin de collaborer le plus facilement possible. C'est pour cette raison que nous avons utilisé l'éditeur de code Visual Studio Code. Bien sûr, en cas de problème difficile à résoudre (voir section 4), nous nous concertions et élaborions une solution ensemble.

4 Difficultés rencontrées

4.1 Traçage des lignes

4.1.1 Géométrie des lignes

Afin de donner un aspect de plan de métro aux lignes de tram, l'affichage des lignes doit se faire uniquement avec des droites horizontales, verticales et obliques (45°). Ce problème, qui nous semblait assez abstrait au premier abord, était finalement relativement simple à résoudre avec une disjonction des cas possibles (en fonction des angles entre deux stations données). Ainsi, nous avons pu déterminer la position intermédiaire à partir des coordonnées centrées des stations et en utilisant de la trigonométrie de base.

4.1.2 Épaisseur des lignes

Plus tard dans le développement, nous nous sommes rendu compte que le traçage de lignes sans épaisseur (fonction « lineRGBA » de l'unité « SDL_GFX ») rendait les lignes peu distinguables du reste. Or la SDL ne supporte pas nativement le traçage des lignes d'une épaisseur donnée. Nous nous sommes penchés sur l'unité « SDL_GFX » qui possède une fonction « thickLineRGBA » permettant de dessiner des lignes avec une épaisseur. Malheureusement, les liens entre l'implémentation C de la SDL et le Free Pascal Compiler étant incomplets, la fonction n'était pas accessible en Free Pascal. Ce souci n'ayant pas été anticipé, nous avons donc dû réfléchir à une solution simple pour contourner le problème. Ainsi, à partir de la fonction « lineRGBA » et de l'angle de la ligne, nous avons dessiné des lignes superposées les unes par rapport aux autres. Il est ainsi possible de créer une ligne épaisse. Dans le cas de lignes verticales et horizontales, il est assez aisé de dessiner une ligne épaisse en incrémentant et décrémentant x ou y autour du centre de la ligne. Cependant, dans le cas de lignes obliques, ce raisonnement n'est pas valide.

4.2 Résolution des itinéraires des passagers

Toute la difficulté du jeu reposant sur la performance du réseau créé, il était impératif pour nous que les passagers fassent preuve d'intelligence dans leur choix d'itinéraire. Dans le cas contraire, il n'y a aucune incitation à construire un plan de réseau efficace. La recherche du plus court chemin est un problème informatique classique, et de nombreux algorithmes existent pour sa résolution. Nous ne sommes pas mathématiciens ni chercheurs, alors nous avons décidé d'en adopter un assez connu : l'algorithme de Dijkstra. Cet algorithme est supposé être de complexité $\mathrm{O}\left((n+a)\,log\,(n)\right)$. Cependant, étant donné que nous avons fait l'implémentation de cet algorithme assez théorique nous-même, en s'inspirant seulement de son fonctionnement quand il est exécuté à la main, il serait très peu surprenant d'apprendre que sa complexité s'en soit trouvée agrandie. L'écriture des fonctions et procédures liées à la résolution d'itinéraire en elle-même n'a pas présenté de difficulté majeure, l'algorithme n'étant pas excessivement compliqué à appréhender. En revanche, toutes les subtilités d'allocation mémoire, les tableaux sur plusieurs dimensions, la rigueur dans le choix des indexes, ou bien la fragmentation des différentes tâches pré-requises à l'exécution de Dijkstra dans de nombreuses fonctions, eux, ont été sources de bugs ayant nécessité de longues heures de débogage. A l'heure d'écrire ce rapport, à de rares occasions, des bugs subsistent toujours et produisent parfois des itinéraires incohérents. Le taux d'occurrence de ces derniers étant faible, la jouabilité est faiblement impactée. Aucune solution n'a été trouvée pour l'heure. De la même manière, lorsqu'elle est confrontée à une correspondance, il arrive que la fonction responsable de la montée des passagers, « Passengers_Get_On », ne fasse pas monter le passager dans le nouveau wagon.

4.3 Animation réaliste des trains

Afin que le mouvement des trains soit réaliste, au lieu d'une vitesse constante, nous avons opté pour une vitesse variable dépendant du temps. Le but étant de mimer le déplacement d'un véhicule réel. On définit alors deux constante : t_a qui est le temps pris par le train pour accélérer, et v_m la vitesse maximale du train. Ainsi, pour le déplacement du véhicule, on peu définir trois périodes :

— L'accélération : entre 0 et t_a, la vitesse du train augmente linéairement jusqu'à atteindre : v_m.

— Croisière : entre t_a et t_d, la vitesse du train est constante et maximale : v_m.

— Décélération : entre t_d et $t_d + t_a$, la vitesse du train diminue linéairement jusqu'à atteindre : v_m .
Soit $t \in [0, t_d + t_a]$ le temps écoulé depuis le départ du train. Avec ces 3 périodes on peut définir la fonction vitesse définie par morceau :

$$v(t) = \begin{cases} v_1(t) = t \times \frac{v_m}{t_a} & \text{si } 0 \leqslant t \leqslant t_a \\ v_2(t) = v_m & \text{si } t_a < t \leqslant t_d \\ v_3(t) = (t - t_d) \times \frac{-v_m}{t_a} + v_m & \text{si } t_d < t \leqslant t_d + t_a \end{cases} , \forall t \in [0, t_d + t_a]$$

Ce qui nous intéresse, c'est d'obtenir la position du train en fonction de t. D'après nos cours de mécanique du point, la vitesse est la dérivée de la distance par le temps. On a donc :

$$v(t) = \frac{d}{dt} d(t) \Leftrightarrow d(t) = \int v(t)\, dt$$

$$d(t) = \begin{cases} d_1(t) = \int_0^t v_1(t) = \frac{t^2}{2t_a} * v_m & \text{si } 0 \leqslant t \leqslant t_a \\ d_2(t) = d_1(t_a) + \int_{t_a}^t v_2(t) = d_1(t_a) + (t - t_a) * v_m & \text{si } t_a < t \leqslant t_d \\ d_3(t) = d_2(t_2) + \int_{t_d}^t v_3(t) = d_2(t_2) + (t - t_d)\, v_m - \frac{v_m t^2}{2t_a} & \text{si } t_d < t \leqslant t_d + t_a \end{cases} , \forall t \in [0, t_d + t_a]$$

Maintenant que nous avons la distance en fonction du temps, il suffit d'implémenter d_1, d_2 et d_3. On utilise des instructions conditionnelles pour déterminer laquelle utiliser en fonction du temps et des domaines de définition de chacune.

4.4 Optimisations

Au fur et à mesure que le projet avançait et que les ressources graphiques se multipliaient. Nous nous sommes heurtés à un problème de performance qui affectait la fluidité du jeu. Ainsi, après quelques recherches, nous avons procédé à des optimisations :

— Tout d'abord, lorsque la SDL crée des surfaces à partir d'images importées, ces dernières ont un format de pixel (profondeur de couleur, masques, canal alpha ...) qui n'est pas forcément le même que celui de l'affichage. Or, lors du rendu avec « SDL_BlitSurface », la SDL doit effectuer une conversion à la volée pour rendre la surface sur l'écran. Cette conversion nécessite beaucoup de temps de calcul. Cependant, il est possible de s'affranchir de cette conversion en convertissant dès l'import des ressources graphiques. Cette conversion est alors effectuée avec la fonction « SDL_DisplayFormat » qui prend pour paramètre la surface à convertir, et renvoie une nouvelle surface, qui est la surface convertie.

— Ensuite, il est possible d'utiliser l'accélération matérielle graphique pour certains calculs graphiques (anti-crénelage, les calculs géométriques, la copie de mémoire...). Pour activer cette fonctionnalité, il faut remplacer l'option « SDL_SWSURFACE » par « SDL_HWSURFACE » dans la fonction qui instancie la fenêtre : « SDL_SetVideoMode ». Cependant, cette optimisation est conditionnée au support de certaines interfaces de programmation (API) graphiques spécifiques au matériel et aux pilotes d'une machine. Ainsi, sur des configurations plus « exotiques » ou anciennes, il se peut que des problèmes de compatibilité empêchent le lancement du jeu.

— Enfin, nous avons minimisé le nombre de calculs graphiques en ne les effectuant que lorsque c'était nécessaire. Par exemple, les étiquettes et chronomètres sont pré-rendus lorsqu'ils sont modifiés ou encore les positions intermédiaires sont pré-calculées lors des modifications des lignes. Le rafraîchissement des graphismes étant appelé très souvent (toutes les 1/60 secondes), il fallait diminuer au maximum les calculs nécessaires à l'affichage.

4.5 Interface graphique

Une interface graphique est nécessaire pour que l'utilisateur puisse interagir avec le jeu. Or, la SDL est une librairie très rudimentaire en termes de fonctionnalités. Ainsi, nous avons créé une abstraction pour les différents éléments de l'interface graphique (boutons, étiquettes, panneau etc) afin d'améliorer la lisibilité du code.

4.6 Manque de temps

Les fonctionnalités obligatoires que nous souhaitions implémenter au début du projet ont pu être réalisées. Cependant, par manque de temps, beaucoup de fonctionnalités optionnelles n'ont pas pu être développées. Par exemple, la fonctionnalité « Chargement/Sauvegarde Partie » présente dans la première analyse descendante a été abandonnée. En effet, la gestion de la mémoire pour ce projet étant surtout dynamique, nous avons estimé que se lancer dans l'implémentation d'un système de sauvegarde de la partie ne serait pas une idée très sage. Le type « arcade » du jeu ne s'y prête de toute manière pas vraiment. Ce manque de temps nous a également empêché de réaliser certaines optimisations dans le code, notamment pour garder en mémoire les itinéraires afin de réduire le nombre de fois où l'on calcule les itinéraires.

5 Conclusion

La réalisation de ce projet nous semblait compliquée voire impossible lorsque le concept nous a été expliqué. Néanmoins, cela nous a poussé à faire preuve d'autonomie et à avoir une bonne organisation afin de respecter notre cahier des charges. C'était une introduction efficace et parfois périlleuse à la gestion de projet mais également un moyen de développer nos compétences en algorithmie et de découvrir certains concepts que nous n'avions pas eu l'occasion d'explorer en cours l'année dernière. Nous avons également rencontré plusieurs difficultés au cours de la réalisation du projet ce qui nous a poussé à faire preuve d'ingéniosité et à faire des recherches pour trouver des solutions adaptées aux problèmes qui sont survenus.

6 Annexes

6.1 Guide d'utilisation

Au lancement du jeu, un menu de démarrage vous est présenté, vous pouvez choisir de quitter ou bien de jouer. Si vous choisissez de jouer, une partie est lancée et des stations et passagers apparaissent. Les passagers sont de la forme de leur station de destination. Chaque station est supposée accueillir 6 passagers, au-delà de ce seuil, un chronomètre est lancé et vous avez alors 40 secondes pour désengorger la station, sans quoi la partie se termine. Pour mener à bien cet objectif, vous disposez d'une ligne sélectionnable par un clic gauche, celle-ci se situe en bas de votre écran et se repère à sa couleur. Pour relier vos stations, sélectionnez la ligne de votre choix, puis, cliquez et déplacez la ligne d'un point de départ à un point d'arrivée. Si vous souhaitez changer le tracé de la ligne sélectionnée, vous n'avez qu'à cliquer sur ce dernier et à le faire glisser sur une station à desservir. Pour supprimer une station d'un tracé, sélectionnez la ligne, puis, cliquez sur la station en question. Vous ne pouvez pas supprimer la première station d'une ligne. Une fois votre tracé effectué, vous pouvez le peupler de trains et de wagons. Au début de la partie, sur la partie gauche de votre écran, vous remarquerez la présence de jetons échangeables contre des locomotives, des wagons ou des tunnels. Le jeton « tunnel » sera directement déduit de votre inventaire lorsque votre ligne traversera une rivière. Pour finaliser la rame, vous devrez vous saisir à l'aide d'un clic gauche du jeton « locomotive » et le glisser-déposer sur le tracé précédemment créé. Cette locomotive dispose d'une capacité de 6 passagers. Si vous souhaitez augmenter ce nombre, vous pouvez glisser-déposer un jeton « wagon » sur cette locomotive, ceci aura pour effet d'ajouter 6 places au train. Au fur et à mesure que le temps s'écoule, à intervalles réguliers, des bonus de fin de semaine vous seront distribués, vous devrez alors choisir lequel vous convient le mieux parmi différents choix (nouvelle ligne, wagon supplémentaire, ...). Enfin, tout au long de la partie, il est possible de mettre le jeu en pause, ou bien de recommencer la partie (en haut à droite de votre écran).

6.2 Analyse descendante

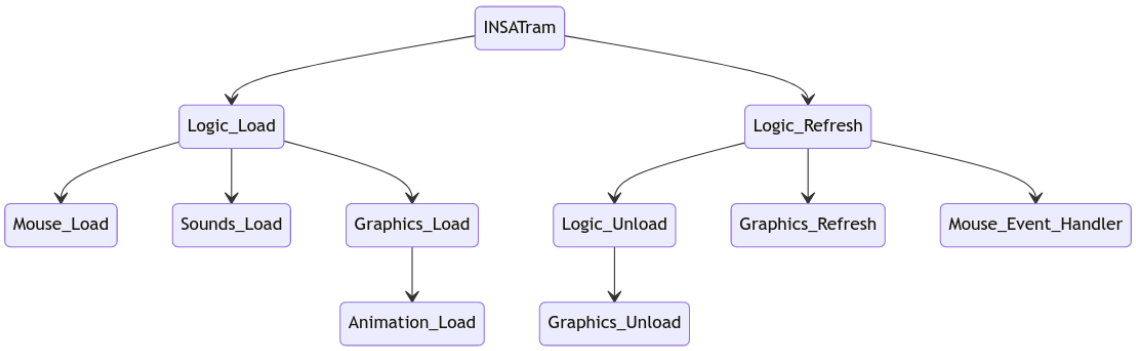

FIGURE 1 – Analyse descendante simplifiée

```
1  Procedure Logic_Load(Var Game : Type_Game);
2
3  Procedure Logic_Refresh(Var Game : Type_Game);
4
5  Procedure Animation_Load(Var Animation : Type_Animation);
6
7  Procedure Animation_Refresh(Var Game : Type_Game);
8
9  Procedure Mouse_Load(Var Mouse : Type_Mouse);
10
11 Procedure Mouse_Event_Handler(Mouse_Event : TSDL_MouseButtonEvent; Var Game :
       Type_Game);
12
13 Procedure Sounds_Load(Var Game : Type_Game);
14
15 Procedure Graphics_Unload(Var Game : Type_Game);
16
17 Procedure Graphics_Refresh(Var Game : Type_Game);
```

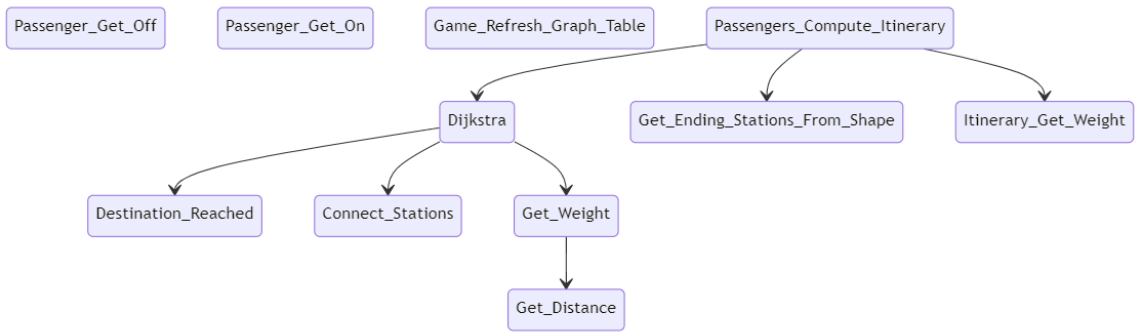

FIGURE 2 – Gestion des itinéraires

```
1  // Fonction qui détermine si le passager doit descendre du train (autrement
       dit, si il a atteint sa destination).
2  Function Passenger_Get_Off(Passenger : Type_Passenger_Pointer; Var
       Next_Station : Type_Station; Var Current_Station : Type_Station) : Boolean
       ;
3
4  // Fonction qui détermine si le passager doit monter dans un train.
5  Function Passenger_Get_On(Passenger : Type_Passenger_Pointer; Var Next_Station
       : Type_Station; Var Current_Station : Type_Station) : Boolean;
6
7  // Procédure qui construit le tableau des liens entres les stations.
8  Procedure Game_Refresh_Graph_Table(Var Game : Type_Game);
9
10 // Procédure qui calcule l'itinéaire des stations correspondant à la forme du
       passager, puis détermine la plus "proche" en prenant l'itinéraire le plus
       court.
11 Procedure Passengers_Compute_Itinerary(Game : Type_Game);
12
13 // Procedure déterminant le plus court chemin entre deux stations données.
14 Procedure Dijkstra(Starting_Station_Index : Integer; Ending_Station_Index :
       Integer; Var Itinerary_Indexes : Type_Itinerary_Indexes; Var
       Reverse_Itinerary_Indexes : Type_Itinerary_Indexes; Game : Type_Game; Var
       Station_Is_Isolated : Boolean);
15
```

```
16 // Fonction qui signale à l'algorithme de Dijkstra que le calcul d'itinéraire
       est arrivé à destination.
17 Function Destination_Reached(Index_Ending_Station : Byte; Dijkstra_Table :
       Type_Dijkstra_Table): Boolean;
18
19 // Procédure qui inscrit dans le tableau de Dijkstra avec quelles stations une
        station donnée peut se connecter.
20 Procedure Connect_Stations(Step : Byte; indexStationToConnect : Byte; Game :
       Type_Game);
21
22 // Fonction qui calcule le poids d'un trajet entre deux stations connectées.
23 Function Get_Weight(Var Station_1, Station_2 : Type_Station): Integer;
24
25 // Procédure qui renvoie toutes les stations qui ont la même forme que le
       passager donné.
26 Procedure Get_Ending_Stations_From_Shape(Game : Type_Game; Passenger :
       Type_Passenger_Pointer; Var Index_Table : Type_Index_Table);
27
28 // Fonction qui calcule le poids (la distance) d'un itinéraire complet.
29 Function Itinerary_Get_Weight(Game : Type_Game; Itinerary_Indexes :
       Type_Itinerary_Indexes) : Integer;
```

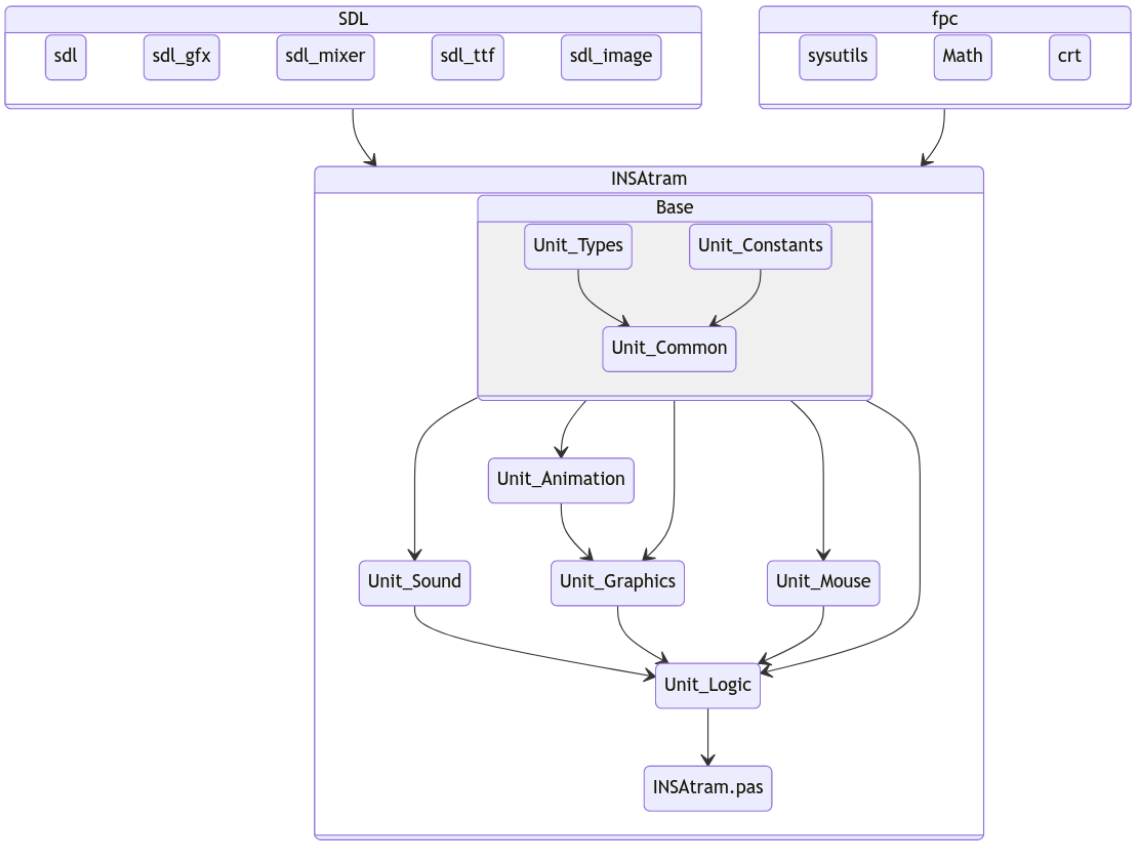

FIGURE 3 – Interdépendances des unités